TP1 – Denotational semantics

Marc CHEVALIER

March 2, 2018

The goal of this session is to program an interpreter to compute the denotational semantics of a simple language.[*]

We will use OCaml, of course. Every version between 4.02.3 and 4.06.1 are ok. Probably older one too, I didn't checked.

1 Language

Start by downloading the package: TP1.tar. It contains:

– a parser for the language, programmed in OCamlLex and Menhir: `frontend/lexer.mll` and `frontend/parser.mly`;

– the type of abstract syntax trees output by the parser, defined in `frontend/abstract_syntax_tree.ml`;

– a pretty-printer, to print back an abstract syntax tree into the original language, in `frontend/abstract_syntax_printer.ml`;

– in `main.ml`, a simple driver that takes a file name passed as argument, parses it, and prints it back.

Typing `make` should compile an executable that runs the simple driver.

1.1 Syntax

The language is a very simple "curly brackets" C-like language. A program is composed of a sequence of statements of the form:

– assignment: `var = expr;`

– tests: `if (expr) stat_1` or `if (expr) stat_1 else stat_2`

– while loops: `while (expr) stat`

[*]Largely similar to `https://www-apr.lip6.fr/~mine/enseignement/l3/2015-2016/denotational.html`. Thanks Antoine!

– blocks in curly brackets `{ stat_1 ... stat_n }`

Non-standard statements include:

– assertions of boolean expressions: `assert (expr);`

– variable printing: `print (var_1,...,var_n);`

– failure: `halt;` which stops the program immediately

Expressions include:

– integer arithmetic operators: `+`, `-`, `*`, `/`, `%` (modulo);

– boolean operators: `&&` (and), `||` (or), `!` (negation);

– integer comparisons `<`, `<=`, `>`, `>=`;

– equality `==` and disequality `!=`, that can be used to compare either two integers or two boolean values;

– constants, including integers, and the boolean constants `true` and `false`;

– the special expression `rand(l,h)` that denotes the non-deterministic interval of integers between the constant `l` and the constant `h`.

The operators have their usual precedence, and you can group expressions using parentheses.

You can use `/* ... */` and `//` comments.

Unlike C, variables do not need to be declared; they start existing when first assigned a value, and keep existing until the end of the program. There are no local variables, and no functions.

1.2 Semantics

Variables have no type and can hold either an integer or a boolean value. Subsequently, we do not distinguish statically between boolean and integer expressions: only values have a type. It is an error to use operators with values of the wrong type, such as adding two boolean values. This is detected at run-time, when evaluating the expression in the current environment.

Other run-time errors include: divisions and modulos by zero; using a non-boolean value in tests, loops and assert conditions; using a variable that has never been assigned to; asserting a condition that is false; executing the `halt` statement.

To avoid having to handle overflows, our integer values will not be represented using machine integers, but as unbounded mathematical integers. You will use the ZARITH library, which provides a simple API (similar to INT32) to manipulate unbounded integers.

2 Deterministic semantics

We first consider the deterministic subset of the language, i.e., we ignore the `AST_int_rand` expression node for now.

Write an interpreter that executes the program by induction on the syntax of statements and expressions; it returns either an environment mapping variables to values, or an error.

You can (and should) use the following steps:

– Define the type `ival` of values. It should contain integers and booleans. Also define the derived type `ival_err` which represents either a correct value, of type `ival`, or an error. You can use a string representation for errors, which will give the user some information on the location and cause of the error. The `ival_err` type will be useful to propagate errors during the evaluation of expressions.

– Define a type `env` for environments. You can use the Map functor from the standard OCaml library to represent mappings from variables to (non-erroneous) values. Likewise, the `env_err` type shall denote either an environment or an error.

– Write an expression evaluation function `eval_expr: env -> expr ext -> ival_err` by induction on the syntax of expressions.

– Write a statement evaluation function `eval_stat: env_err -> stat ext -> env_err`. When should the function return an error environment?

– Test your interpreter on the programs from the `examples` directory. Can you detect infinite loops in `loop.c`, `loop2.c`, and `loop3.c`? Under which condition does your interpreter terminate?

3 Non-deterministic semantics

We now consider the full language including the non-deterministic expression node `rand(l,h)`.

Write an interpreter for this language that outputs the set of all possible environments at the end of the program as well as the set of all errors that can be encountered.

The structure of the interpreter will be similar to the one in the previous question. You can use the following steps:

– Define a type `ival_err_set` to represent sets of `ival_err` objects, i.e., sets containing values and errors. You can use OCaml's standard Set functor.

– Define a type `env_err_set` to represent sets of environments and errors.

– Program a function `eval_expr: env -> expr ext -> ival_err_set` to evaluate an expression in a single environment and return the set of its possible values (and errors) in that environment. When encountering a unary node, the operator must be applied to each possible value of its argument expression; you can use iterators such as fold. Binary nodes require nested fold.

– Program a filter function filter: `env_err_set -> expr ext -> env_err_set` that returns the subset of its `env_err_set` argument that can satisfy the expression, enriched with the errors encountered during the expression evaluation. This function will be useful to model loops, tests and assertions. Remember that an environment can satisfy both an expression and its negation!

– Program a generic fixpoint operator `fix: ('a -> 'a) -> 'a -> 'a` that iterates a function from a base element to reach a fixpoint. Use it then in the semantics of loops.

– Test your interpreter on the `examples` directory, including non-deterministic programs such as `gcd_nd.c` and `loop4.c`.

4 Extensions

Here are a few possible extensions you can implement in the language:

Uninitialized This extension adds a notion of "uninitialized" value. If a variable is used before it is assigned a value, the executions continues by returning the "uninitialized" value. The "uninitialized" value is propagated by all operations ("uninitialized"+1 equals "uninitialized").

Machine integers This extension changes the semantics of the integer data-type so that 32-bit machine integers are used instead of unbounded integers. You can design a version where overflows result in a wrap-around, following two's complement arithmetic, or a version where overflows cause run-time errors.

5 Testing infrastructure

The package provide a ad-hoc testing infrastructure.
It checks output, exit code and termination (not really).
So, if you want to use it, there are a few constraints:

– If the program terminate with at least one error(including `halt`), the exit code should be 1, 0 otherwise. You may be interested in `val exit: int -> 'a` of `Pervasives` standard module.

– Each `print` statement should output exactly one line. Details below

For the deterministic interpreter:

– The output of the statement `print(a, b, c)`, assuming a=1, b=42 and c=true, should be the following JSON value: `{"a":1,"b":42,"c":true}`. Note that you have to escape the double quotes.

– Run the tester with `./tests.py --deterministic`.

For the non deterministic interpreter:

– Considering 3 possibles environments: a=1, b=12; a=2, b=12 and a=2, b=11, the output of the statement `print(a, b)` should be the following JSON value: `[{"a":1,"b":12}, {"a":2,"b":12}, {"a":2,"b":11}]`. Again, remember to escape. There is no order required.

– Run the tester with `./tests.py`. NB: it will run the deterministic test cases, too. The environment is expected to be a singleton.

– Test cases should be such that there is no concurrent printing.